UNIVERSITY OF ALBERTA

Context Sensitivity

CMPUT 416/500
Foundations of Program Analysis

Karim Ali
@karimhamdanali

Previously

- Inter-Procedural Data-Flow
- Inherited vs Synthesized Analysis Info
- Caller-Callee Relationships
- Valid/Invalid Paths
- Staircase of Calls and Returns
- Demand-Driven Analysis
- Types of Contexts
- Important Language Features

How can we conduct context-sensitive analyses?

Method Cloning

Method Cloning

```
int i = 0;
int j = inc(i);
int k = inc(j);
```

Method Cloning

```
int inc(x){
  int y = x+1;
  return y;
}
```

```
int i = 0;
int j = inc(i);
int k = inc(j);
```

Method Cloning

```
int inc1(x){
  int y = x+1;
  return y;
}

int inc2(x){
  int y = x+1;
  return y;
}
```

```
int i = 0;
int j = inc1(i);
int k = inc2(j);
```

Method Cloning

- ❌ Inefficient: too many copies for real programs
- ❌ Expensive operation
- ❌ Recursion!
- ✅ Yet simple and easy to understand

```
int inc1(x){
  int y = x+1;
  return y;
}

int inc2(x){
  int y = x+1;
  return y;
}

int i = 0;
int j = inc1(i);
int k = inc2(j);
```

Method Inlining

Method Inlining

```
int i = 0;
int j = inc(i);
int k = inc(j);
```

Method Inlining

```
int inc(x){
  int y = x+1;
  return y;
}
```

```
int i = 0;
int j = inc(i);
int k = inc(j);
```

Method Inlining

```
int inc(x){
  int y = x+1;
  return y;
}
```

```
int i = 0;

int x1 = i;
int y1 = x1+1;
int j = y1;

int k = inc(j);
```

Method Inlining

```
int i = 0;

int x1 = i;
int y1 = x1+1;
int j = y1;

int x2 = j;
int y2 = x2+1;
int k = y2;
```

Method Inlining

- ✗ Lost procedure abstraction
- ✗ Exponential blow-up
- ✗ Recursion!
- ✓ One procedure => intra-procedural

```
int i = 0;

int x1 = i;
int y1 = x1+1;
int j = y1;

int x2 = j;
int y2 = x2+1;
int k = y2;
```

… so what do we do?

Context Sensitivity

Context Sensitivity

Call Strings

Functional

Context Sensitivity

Call Strings

Functional

- Extend facts with context strings
- Re-evaluate procedure for each extension
- ✅ Universally applicable
- ❌ Recursion!

Context Sensitivity

Call Strings

- Extend facts with context strings
- Re-evaluate procedure for each extension
- ✅ Universally applicable
- ❌ Recursion!

Functional

- Compute summary function per callee
- Apply summary to each context
- ✅ Recursion!
- ❌ Not always applicable

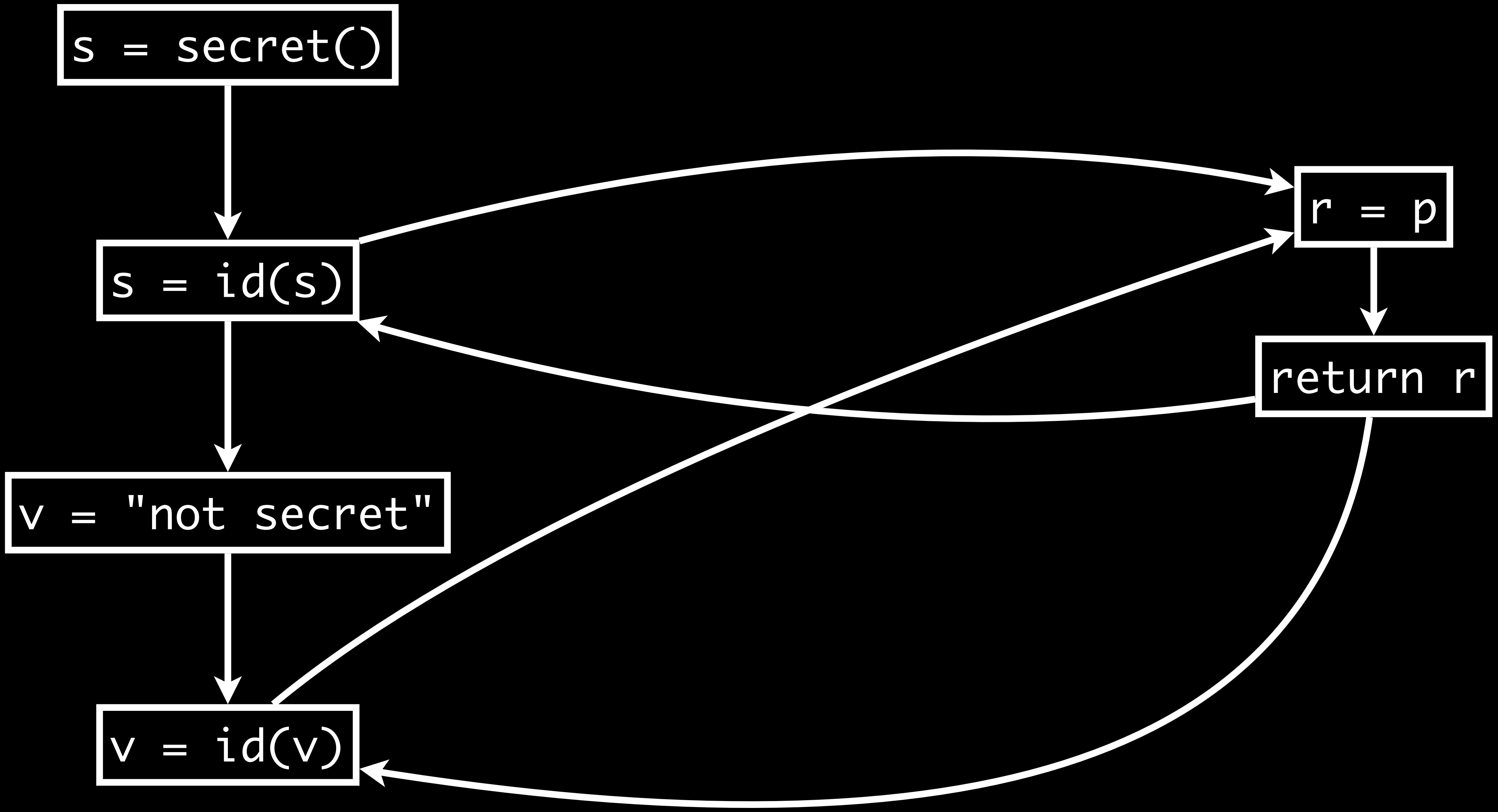
Call Strings

Call Strings

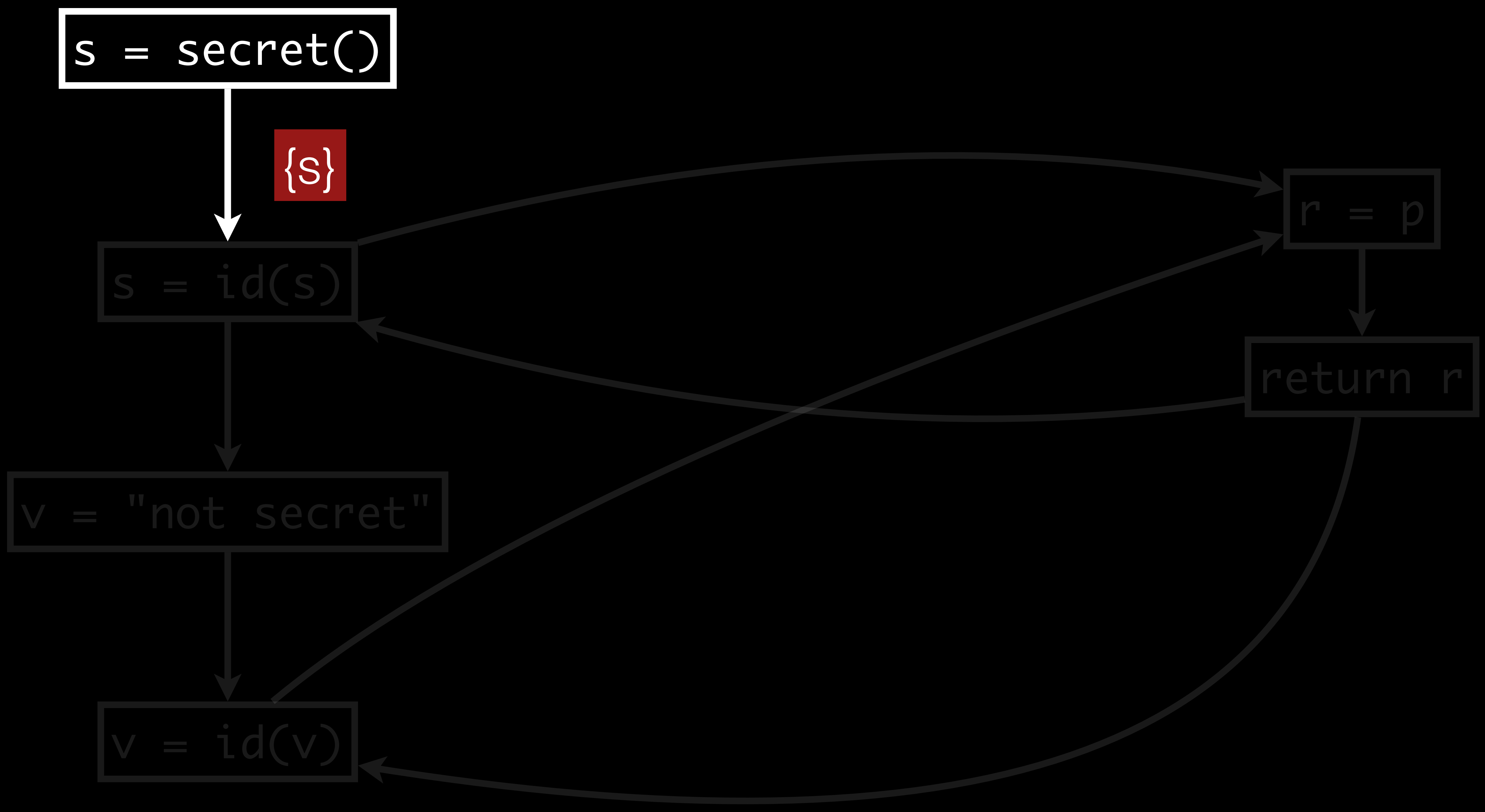
```
main(){
  s = secret();
  s = id(s);
  v = "not secret";
  v = id(v);
}
```

```
id(p){
  r = p;
  return r;
}
```

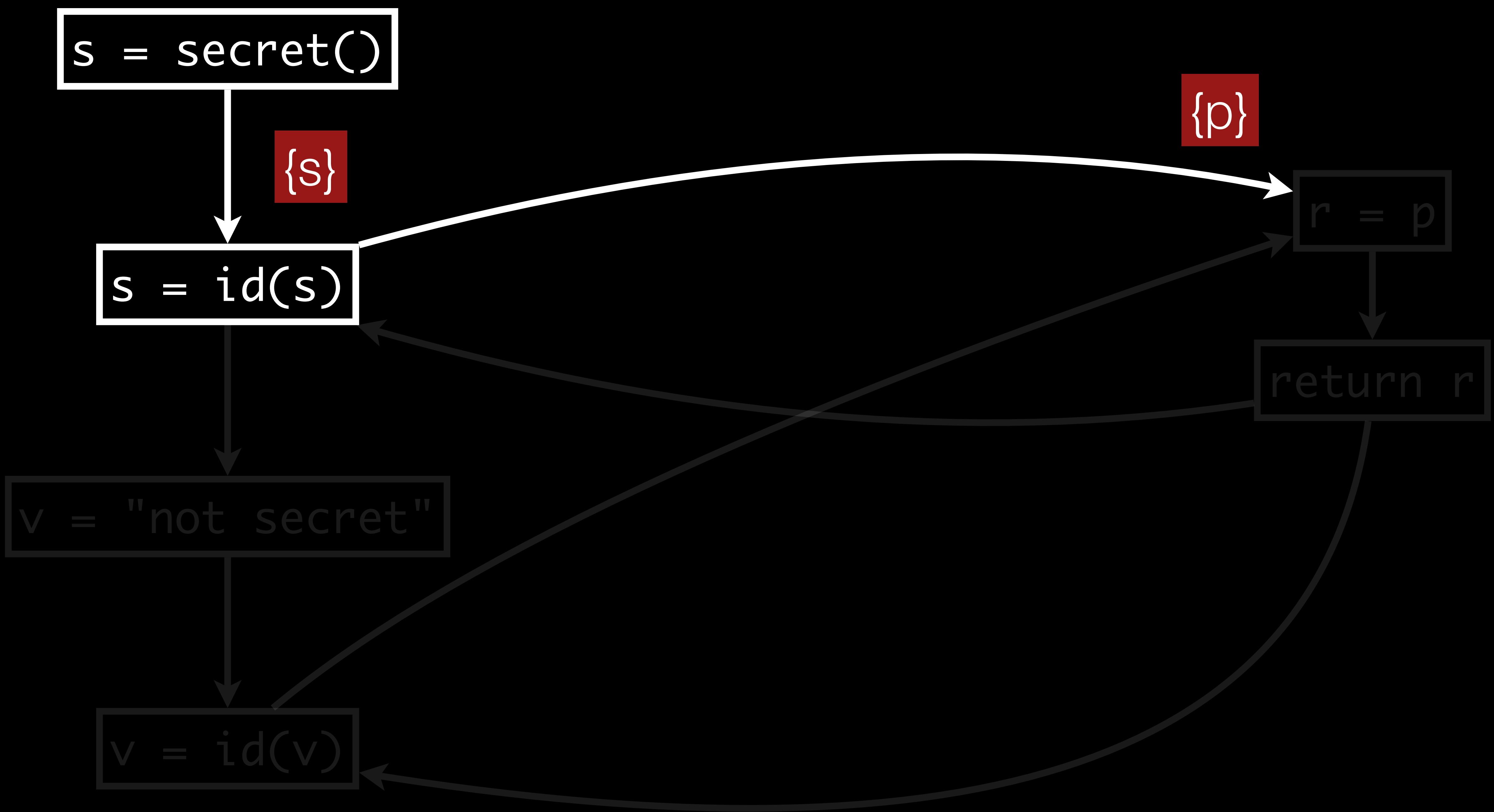
Call Strings



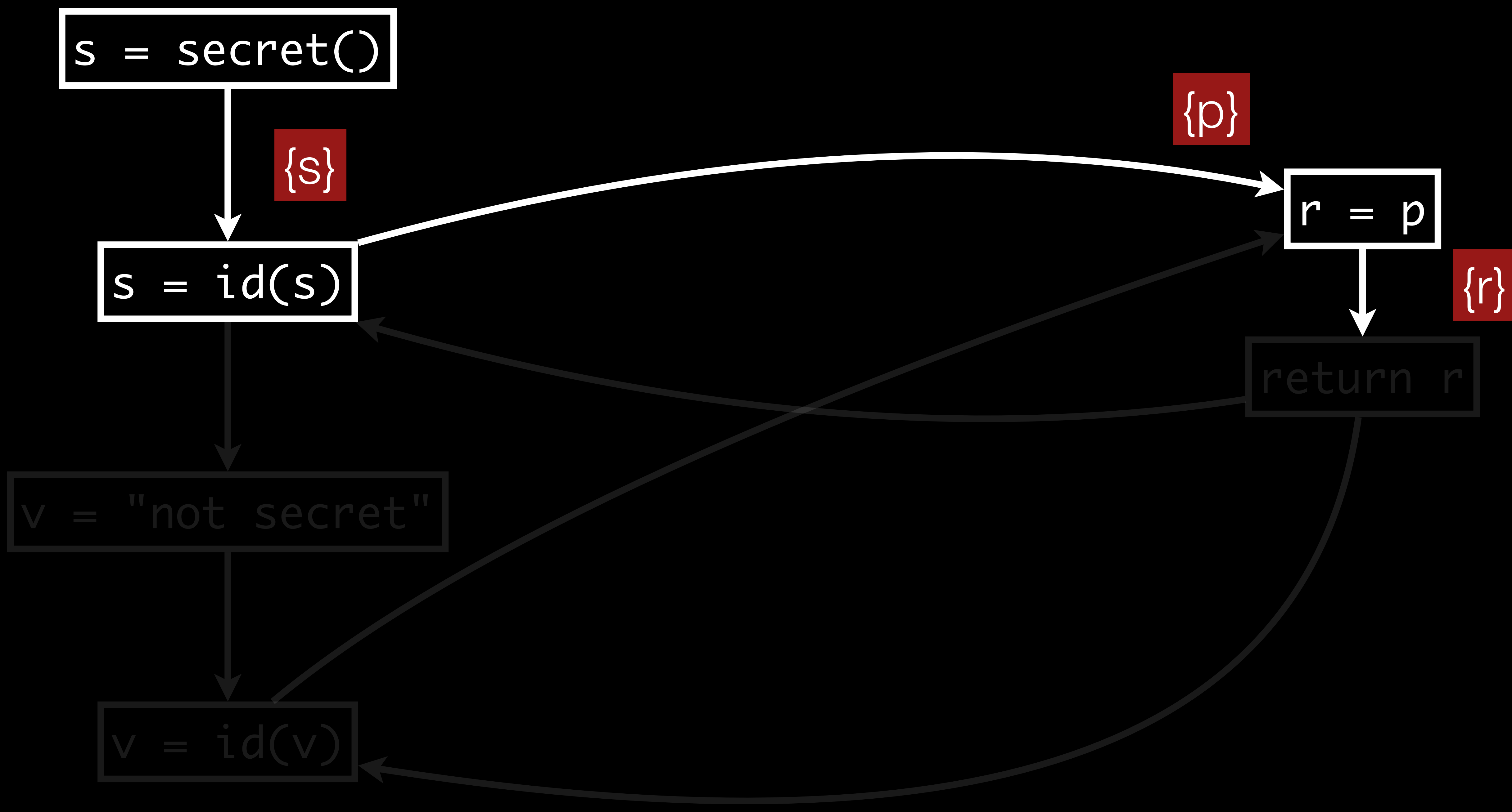
Call Strings



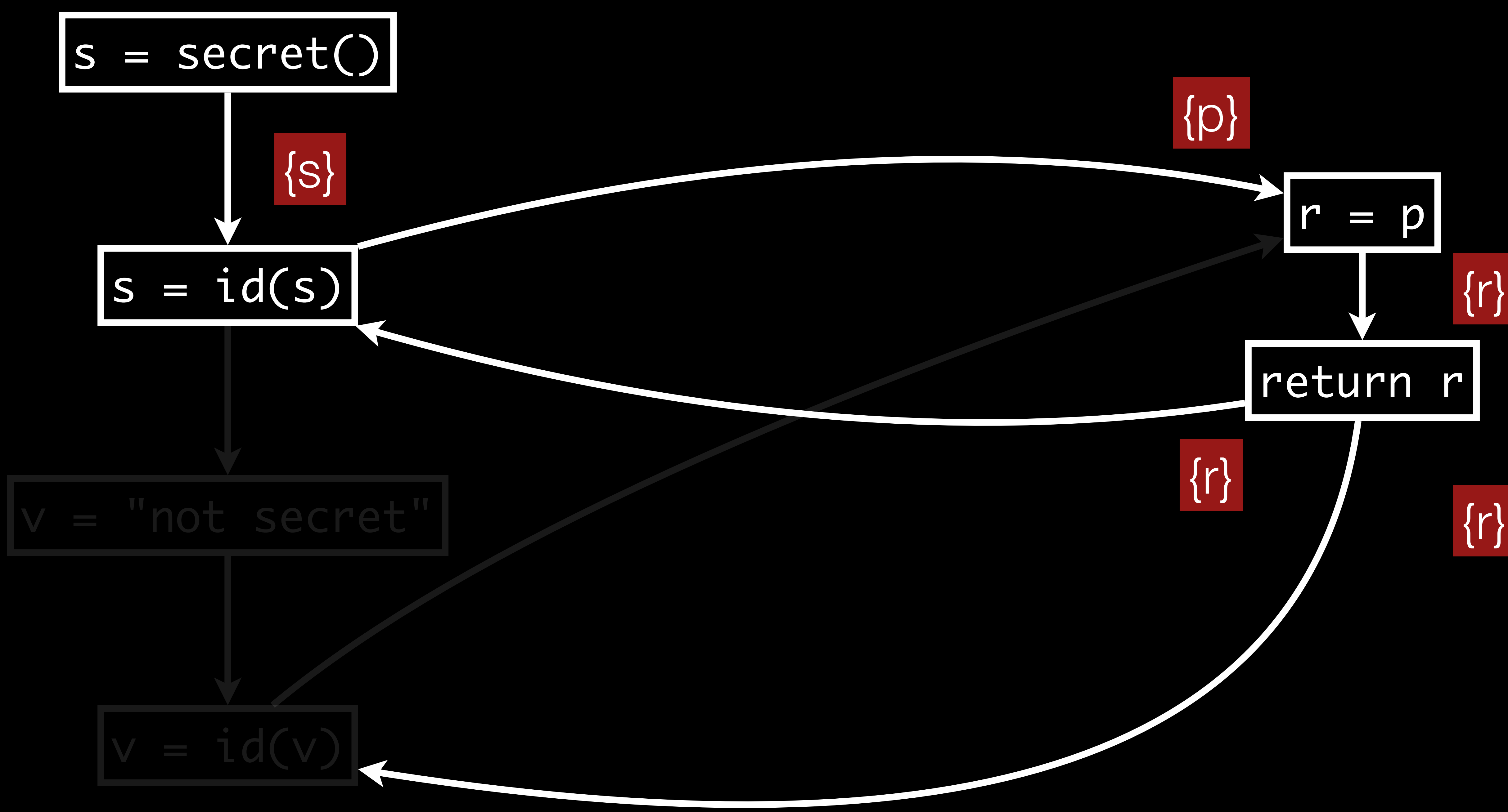
Call Strings



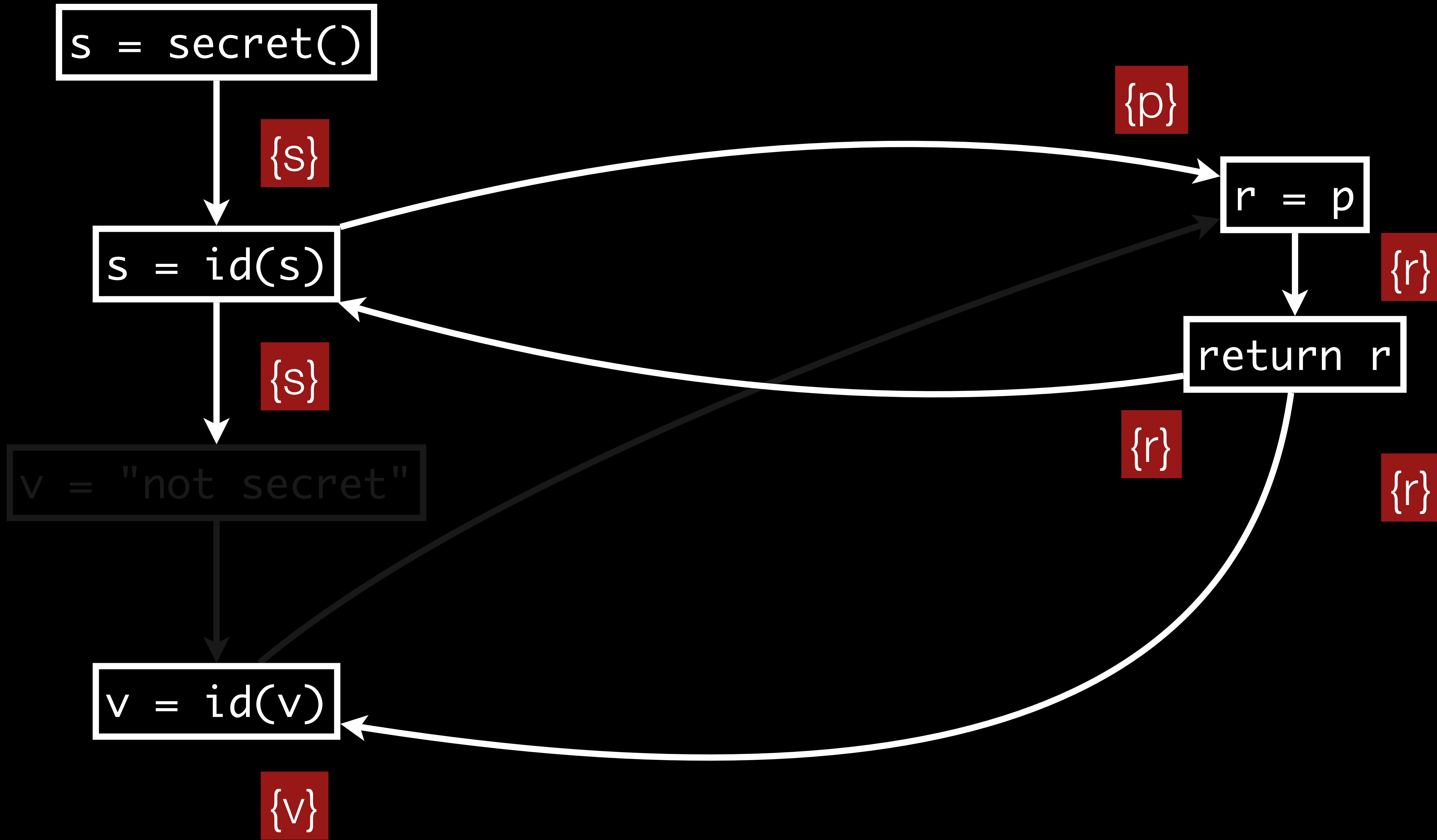
Call Strings



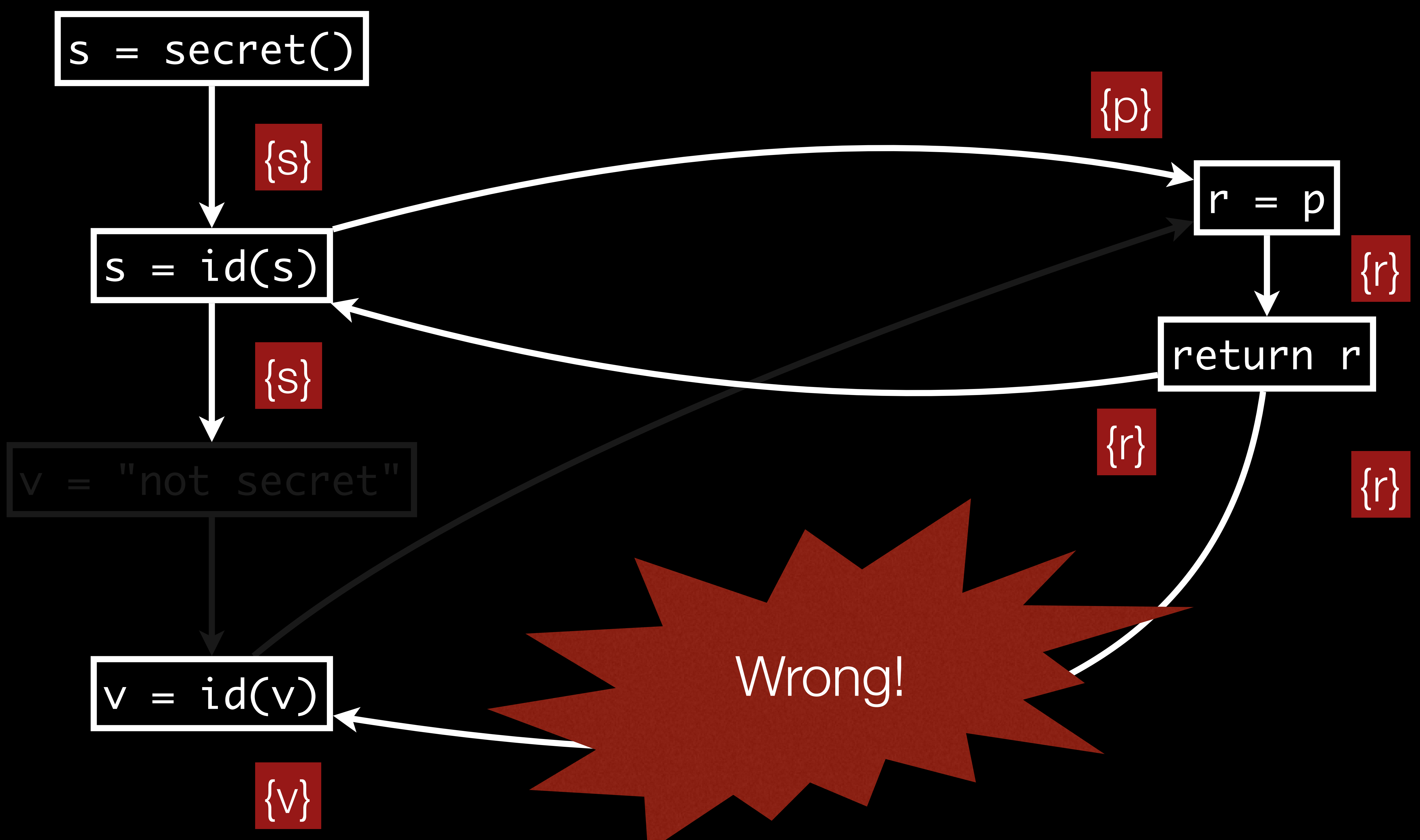
Call Strings



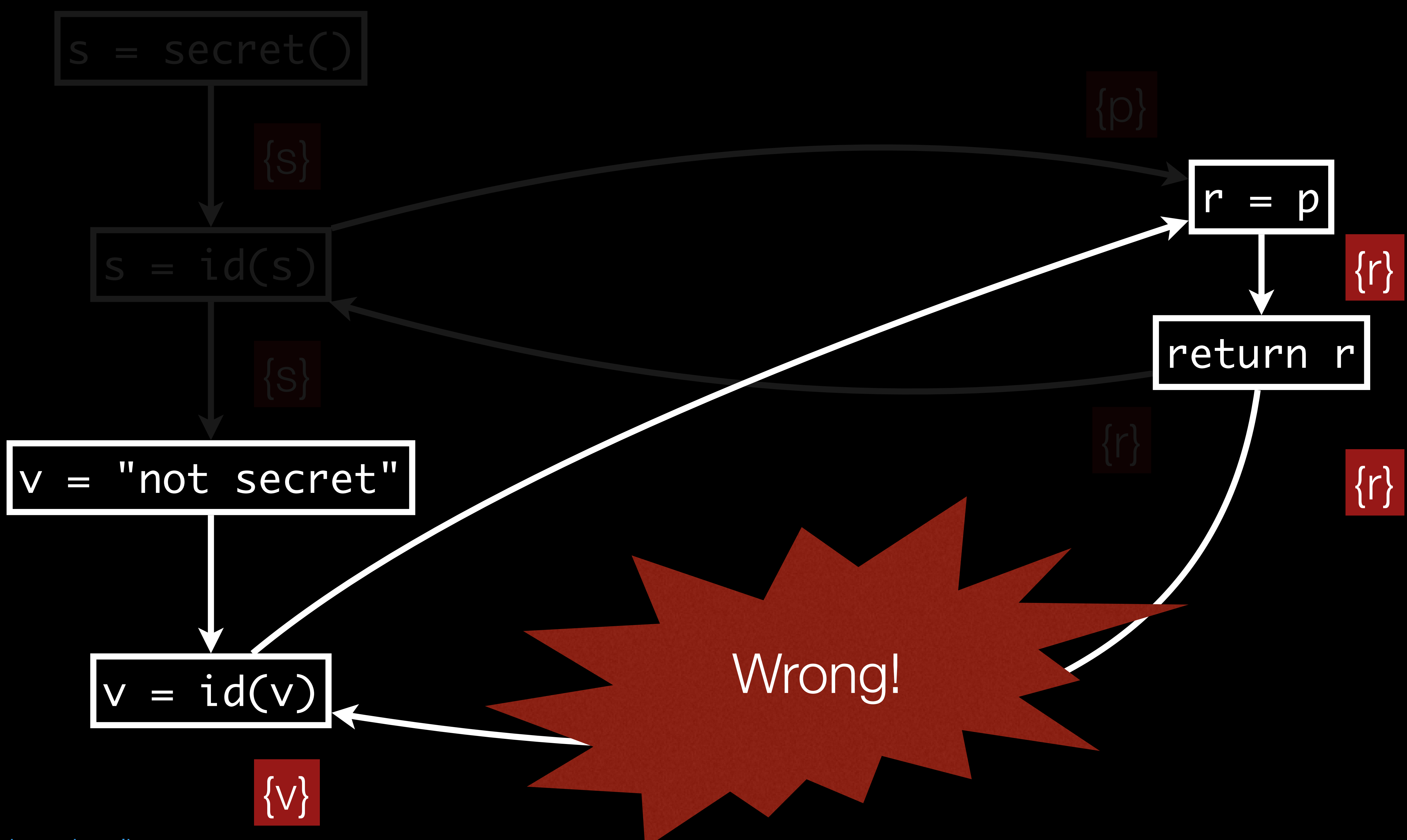
Call Strings



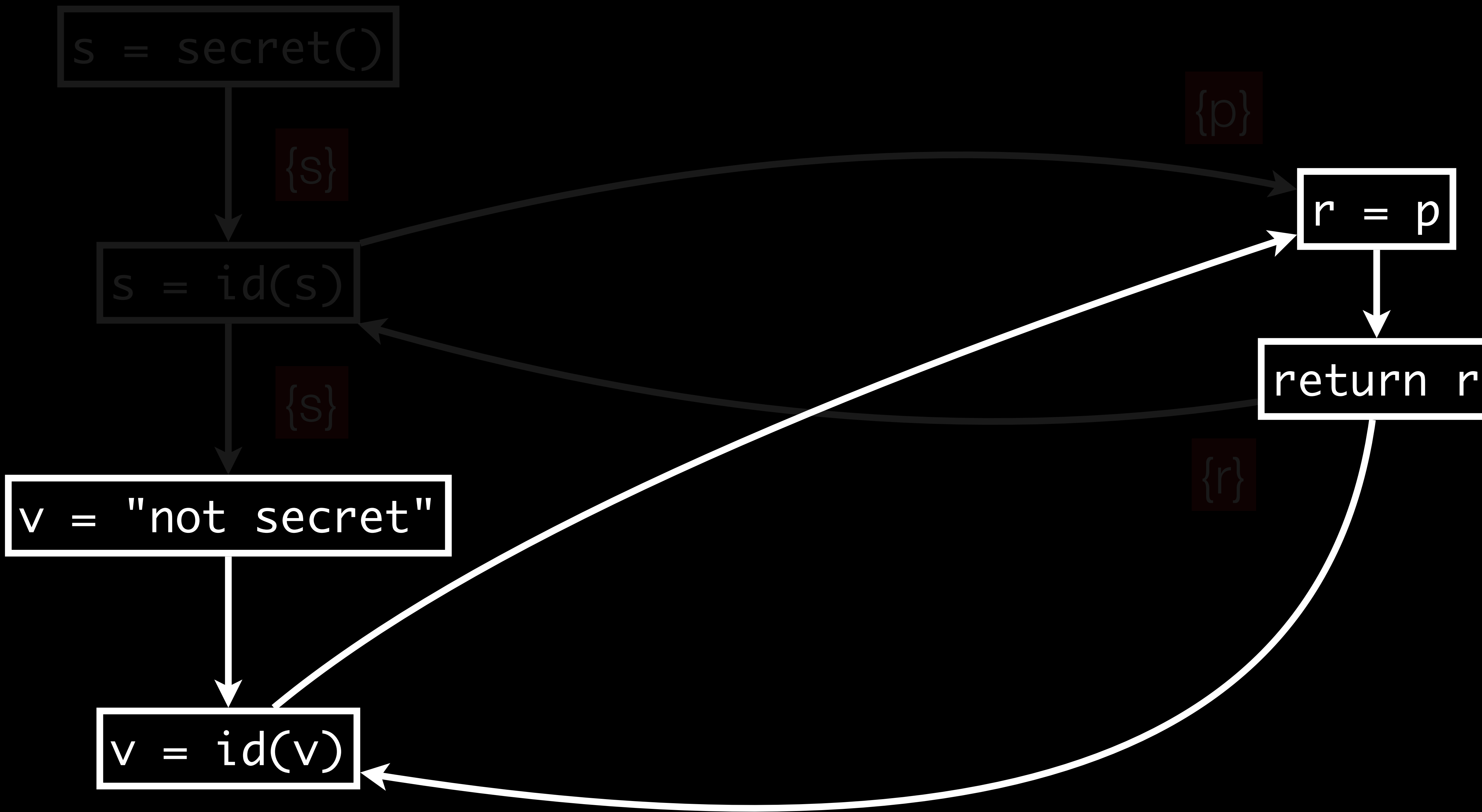
Call Strings



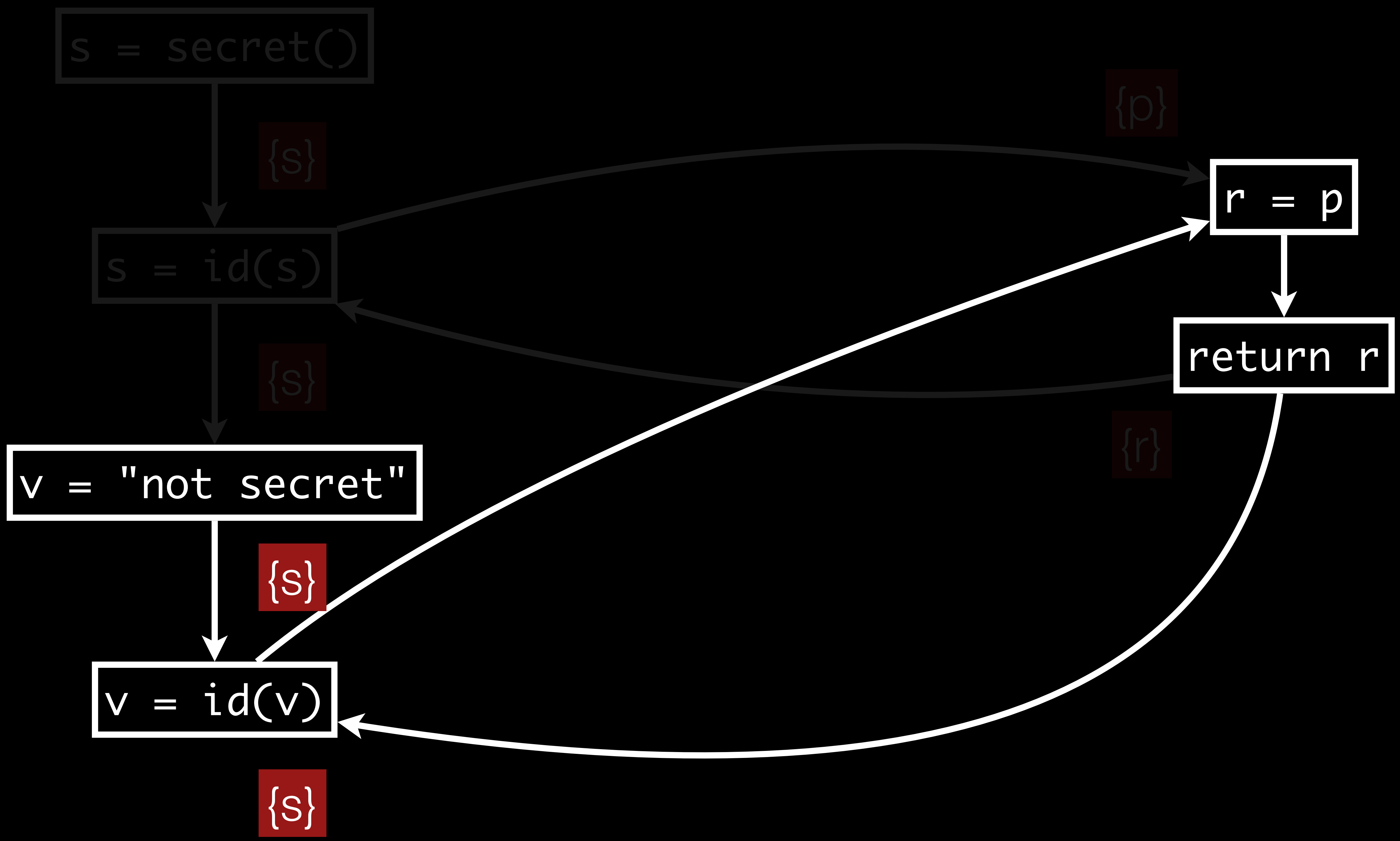
Call Strings



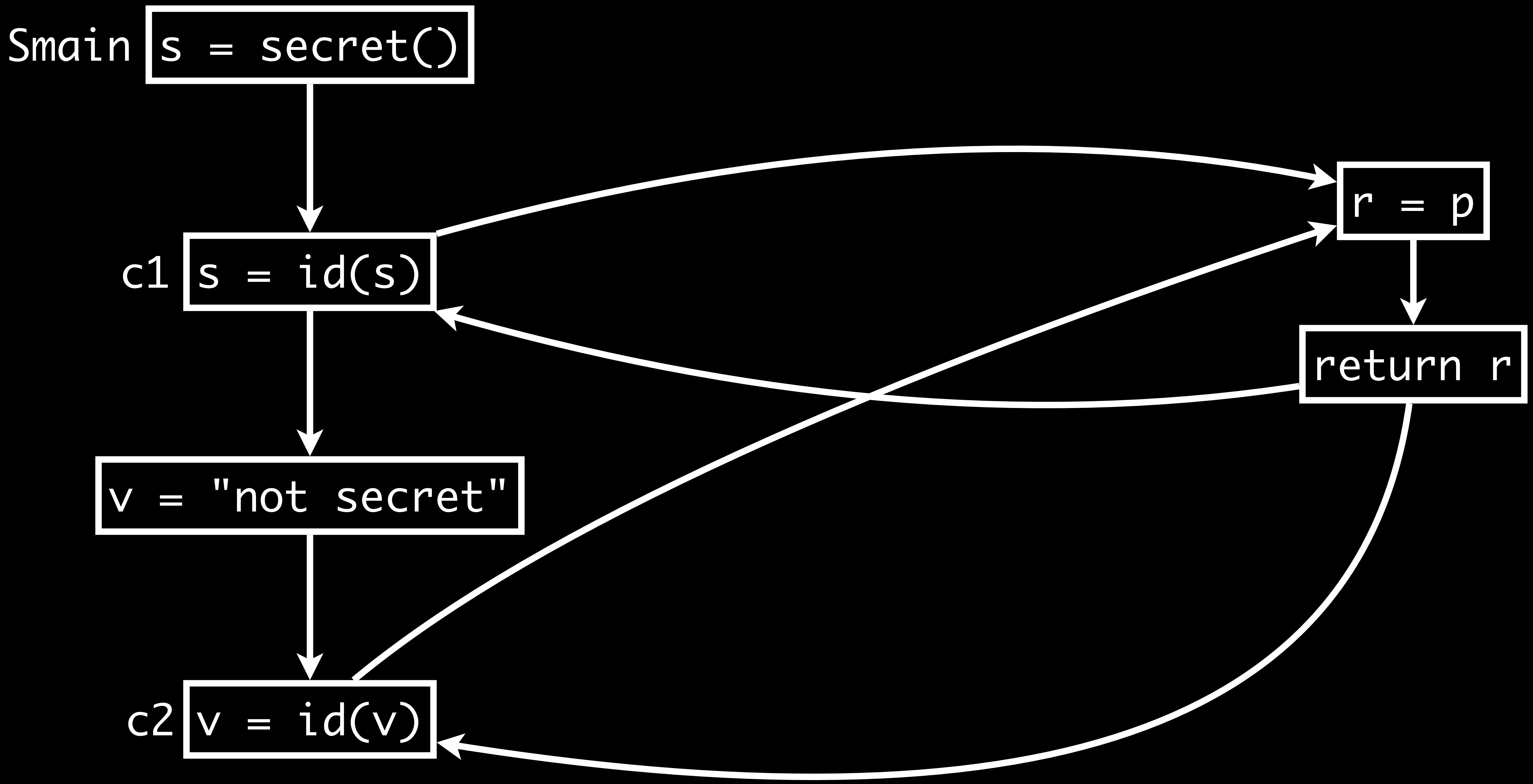
Call Strings



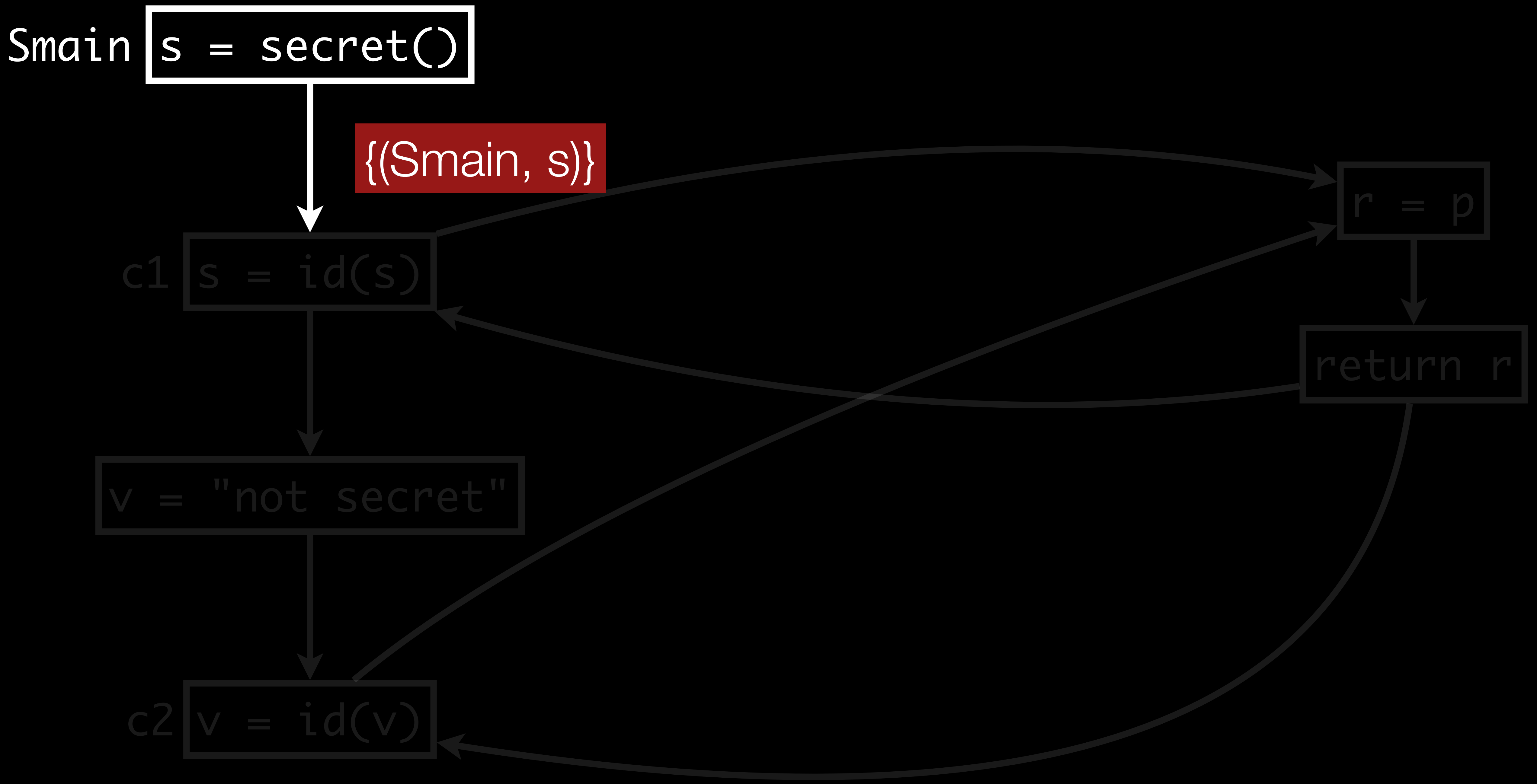
Call Strings



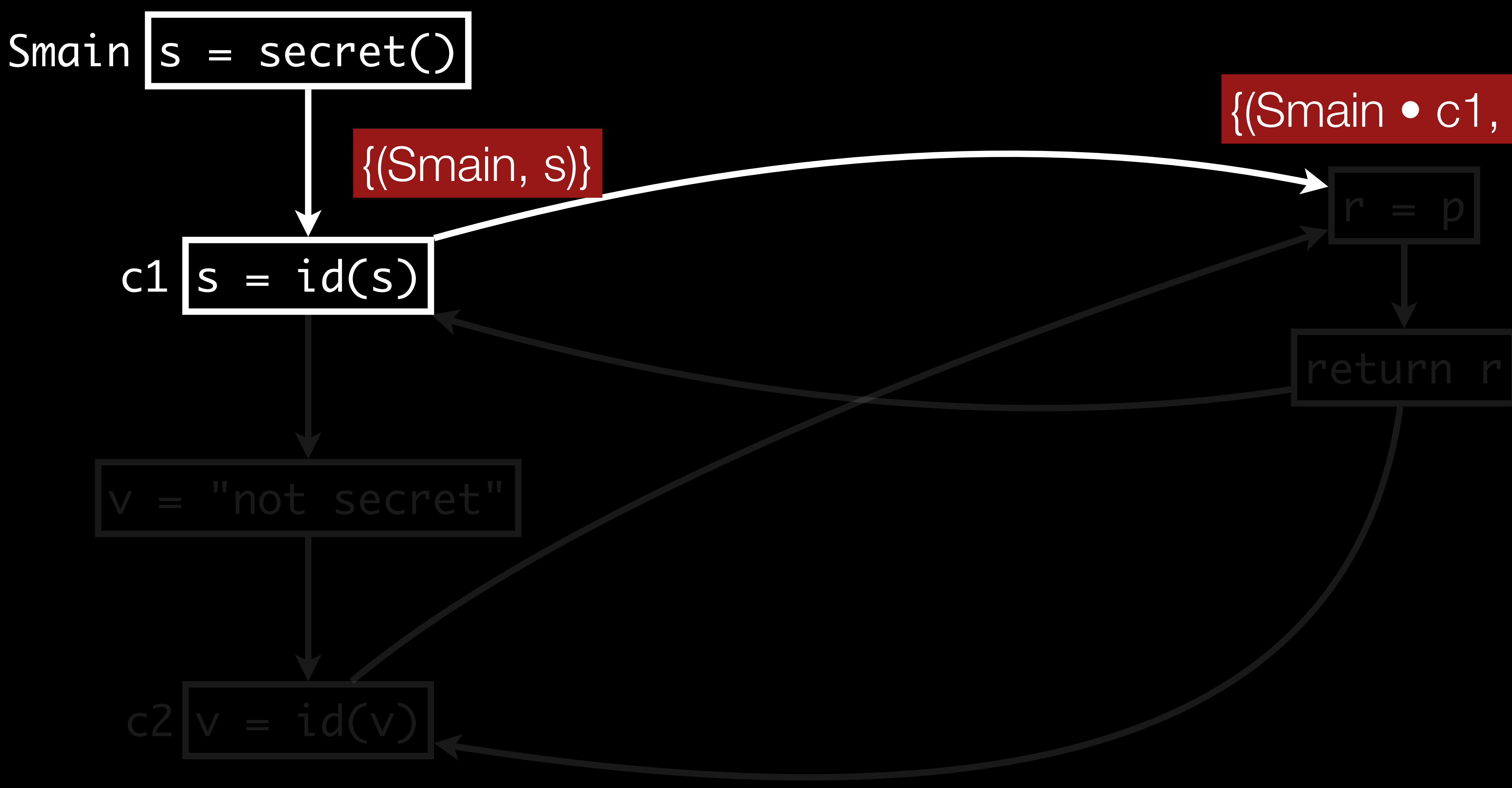
Call Strings



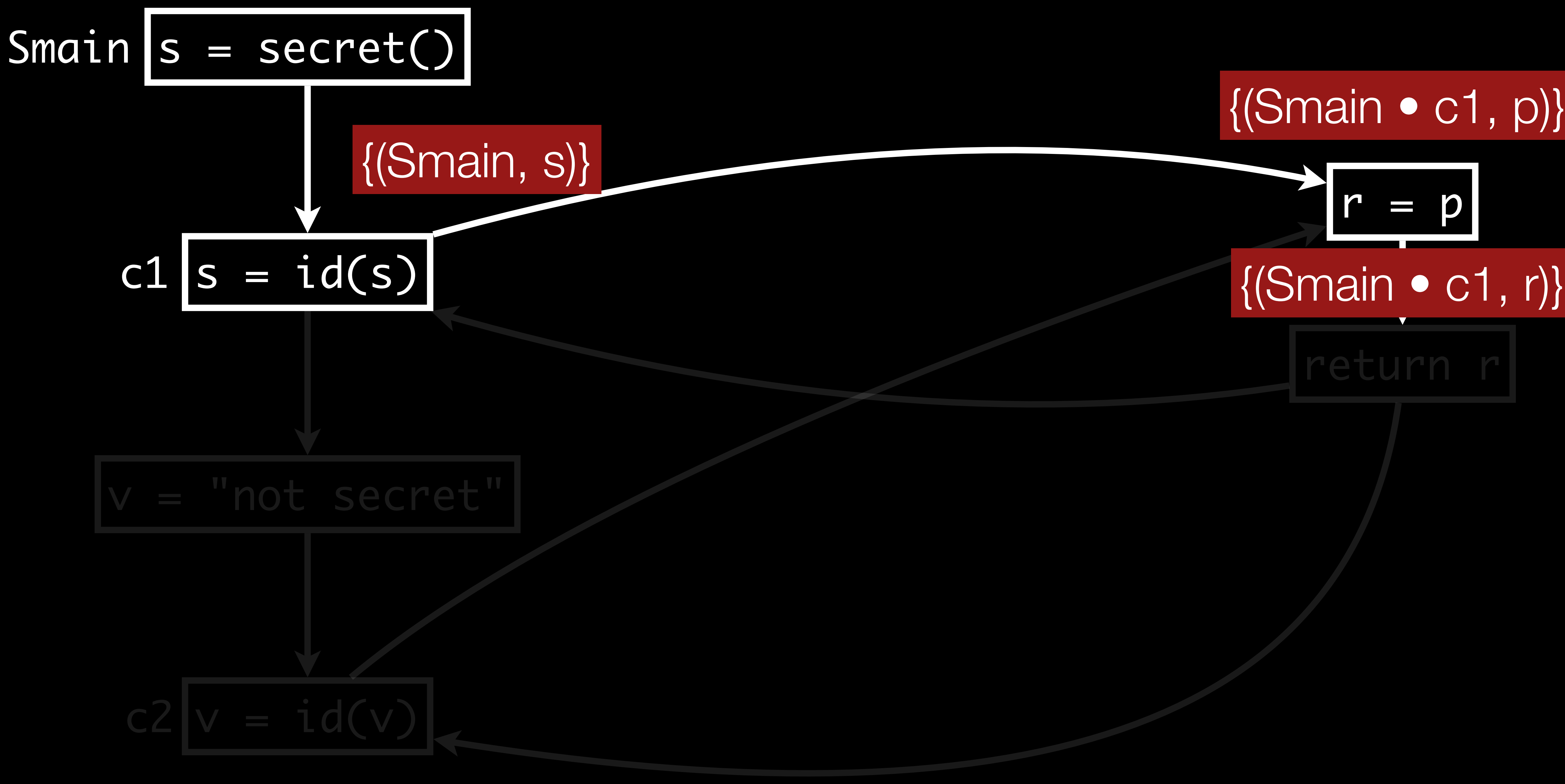
Call Strings



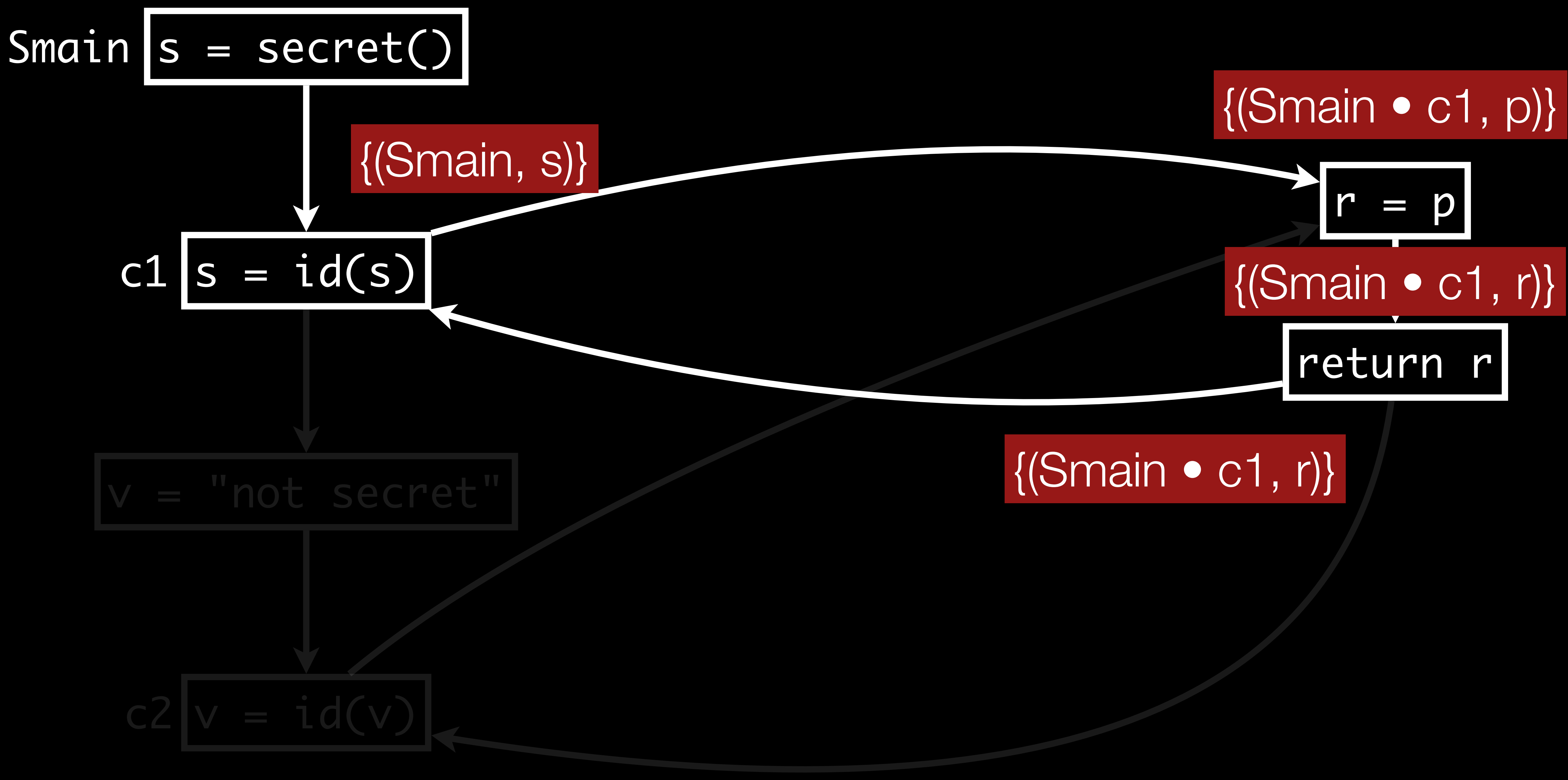
Call Strings



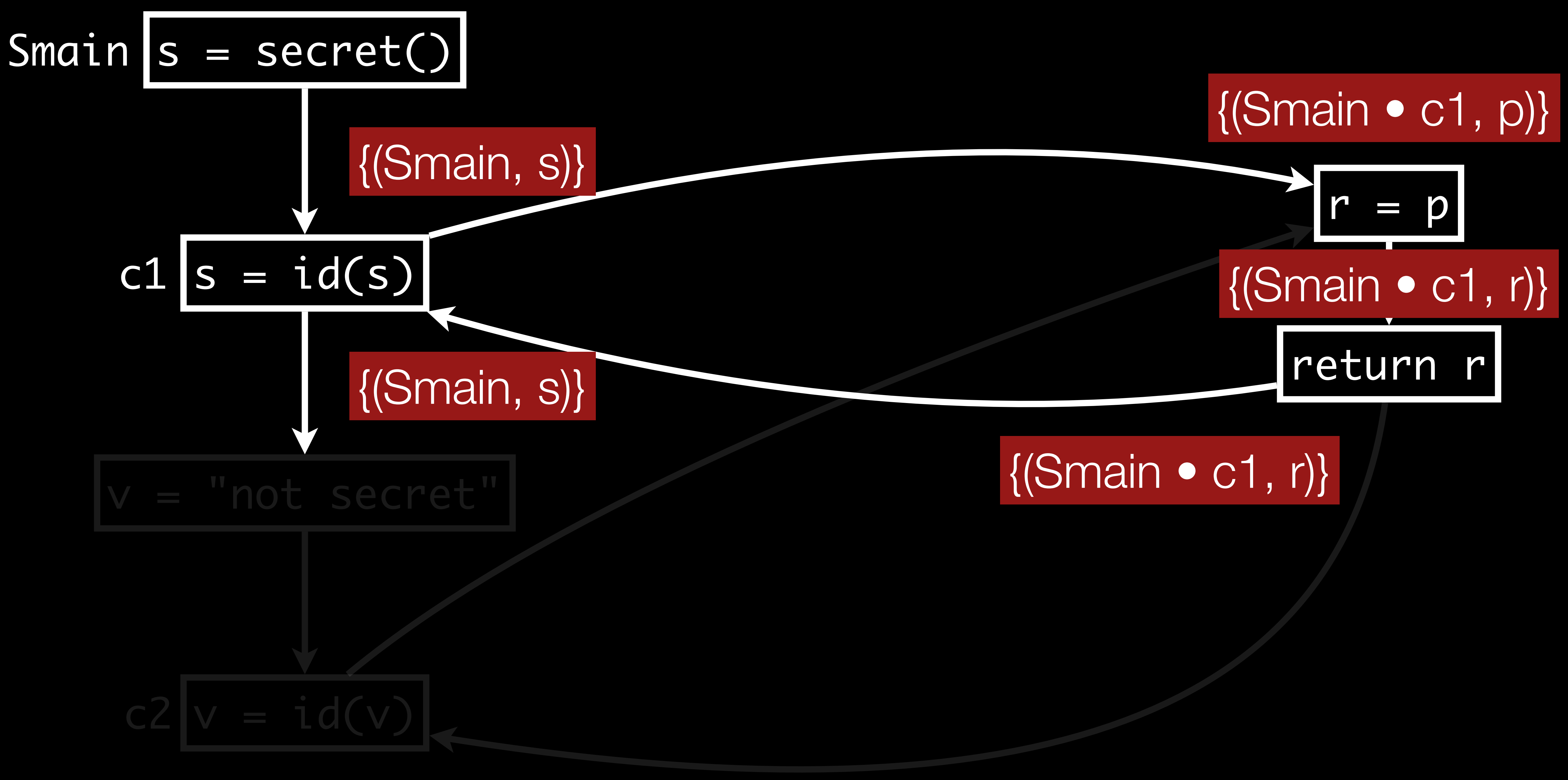
Call Strings



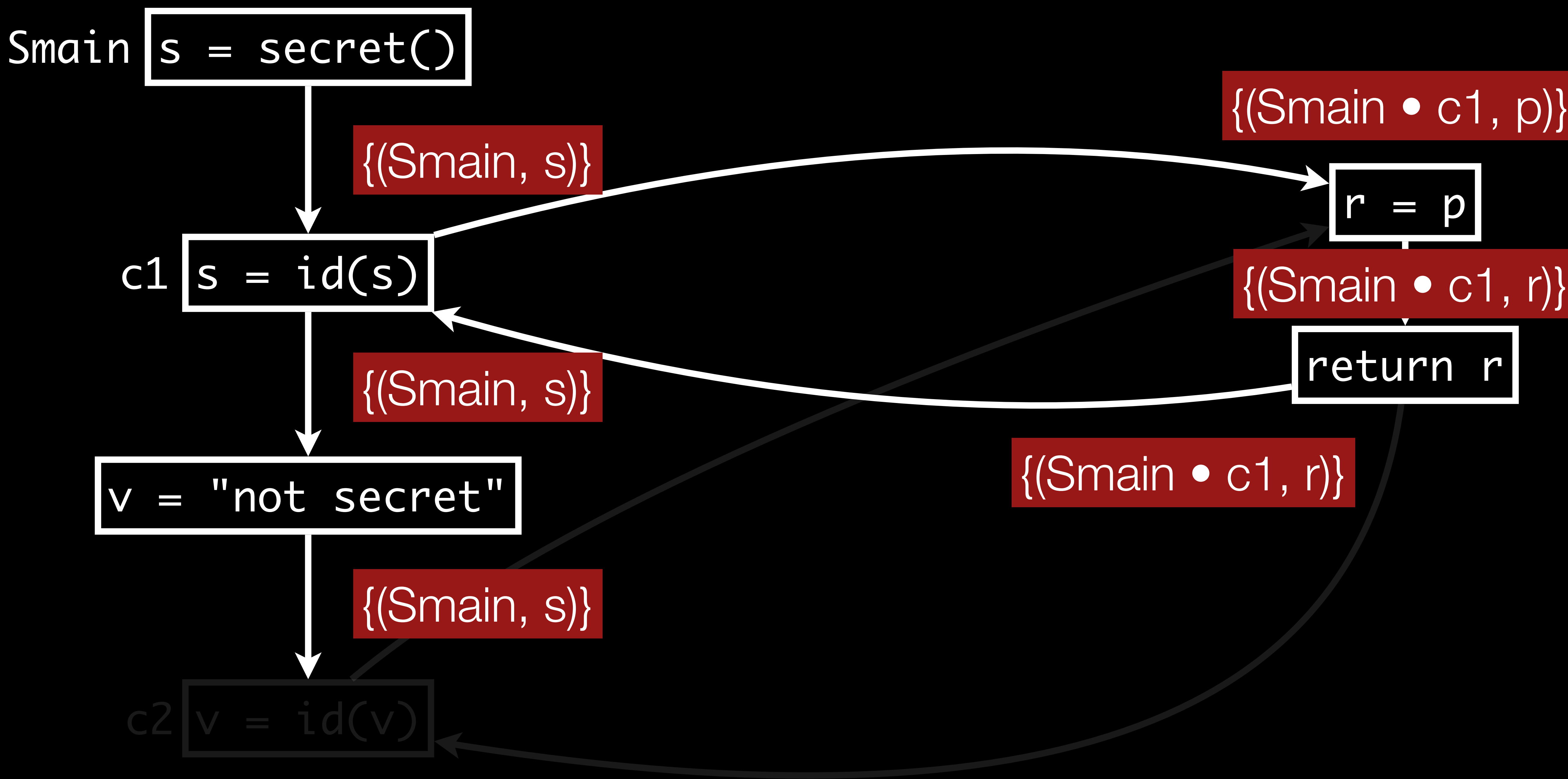
Call Strings



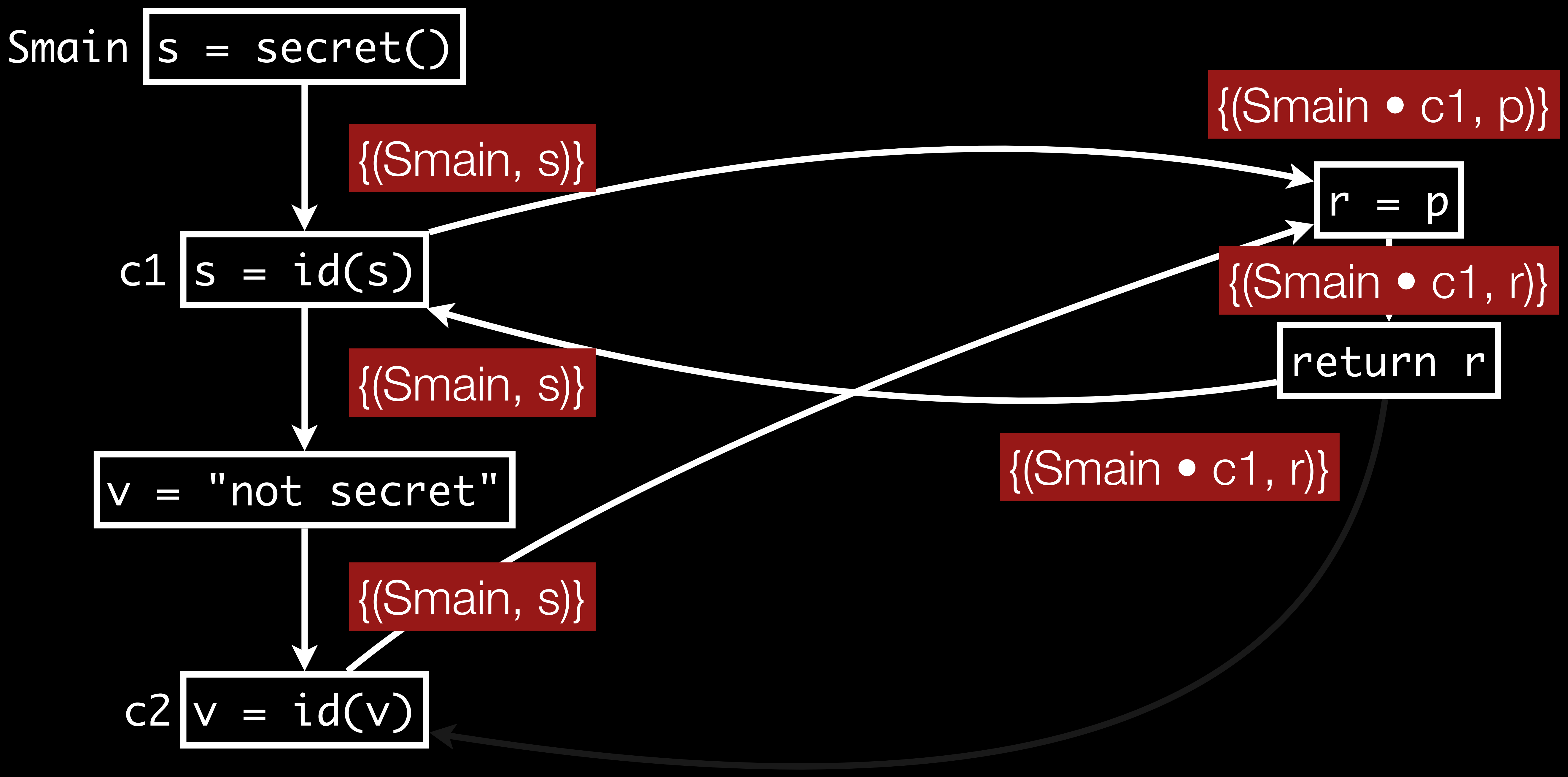
Call Strings



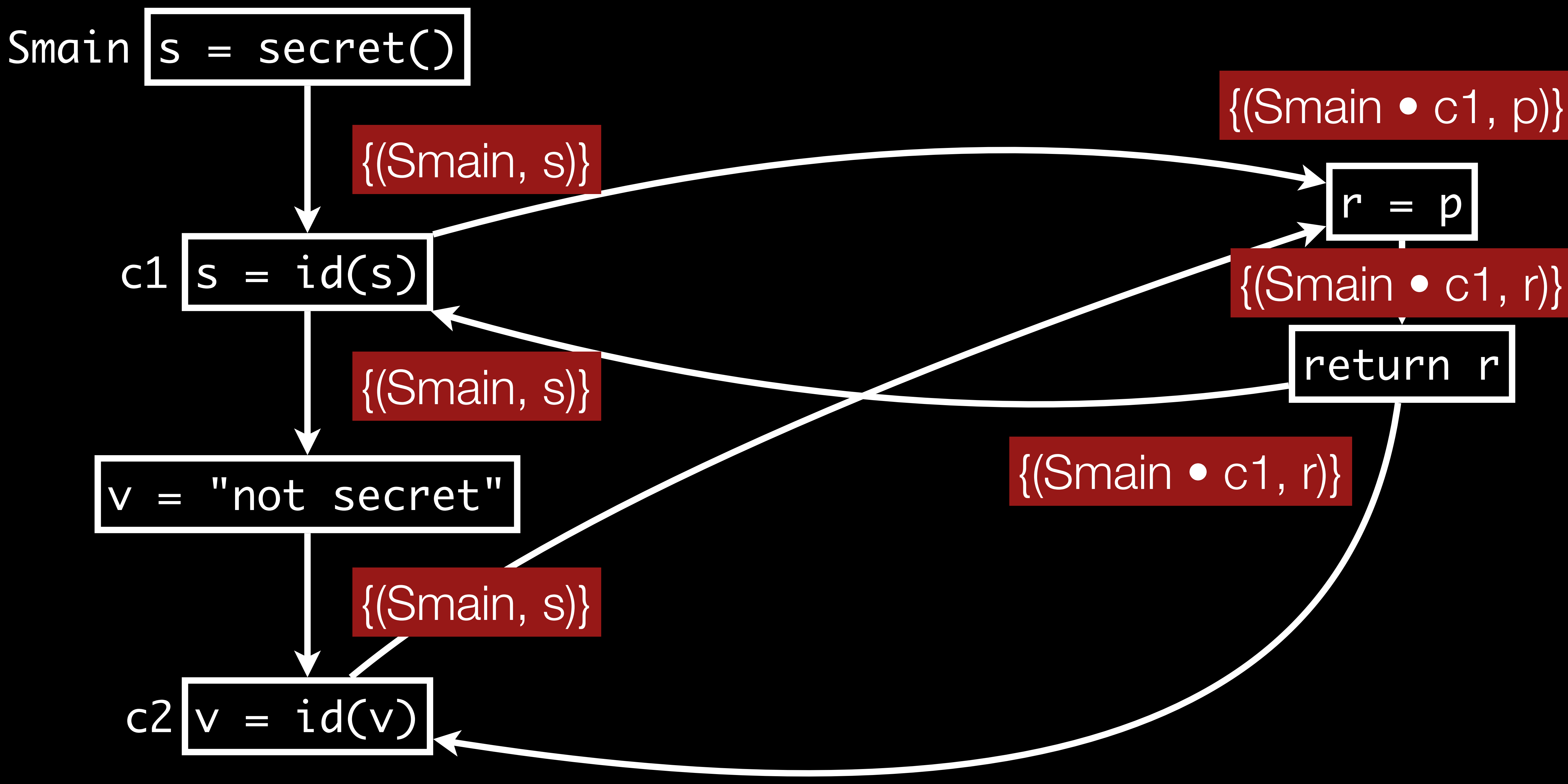
Call Strings



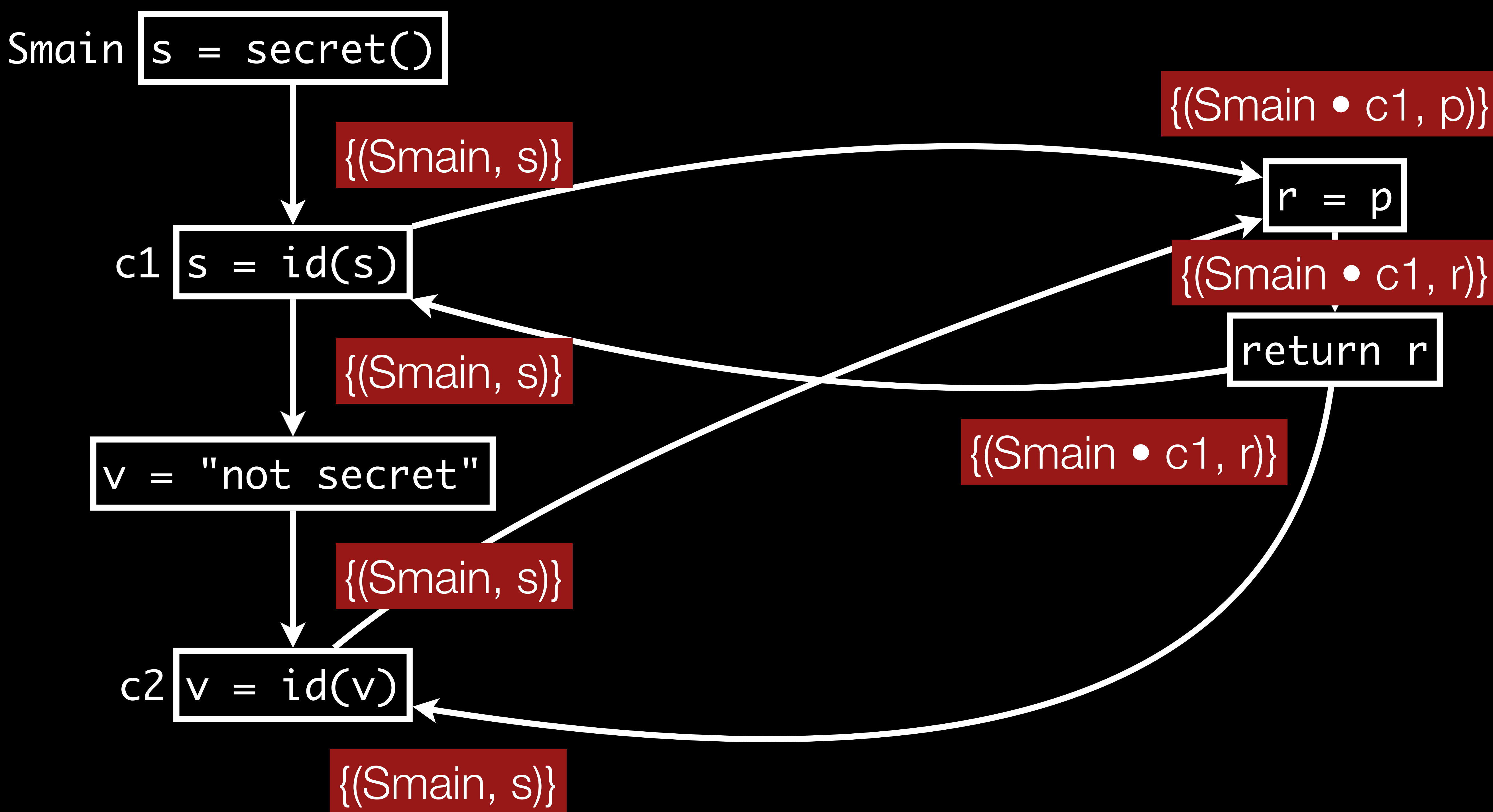
Call Strings



Call Strings



Call Strings



Call Strings
General Algorithm

- Call edges push call site to the stack of calling contexts of each propagated value
- Return edges return only to method on top of the stack of calling contexts
- Handle merge points

Call Strings
Merging Facts

$$X \uplus Y = \{ < \sigma, x \sqcup y > \mid < \sigma, x > \in X, < \sigma, y > \in Y \} \cup$$

$$\{ < \sigma, x > \mid < \sigma, x > \in X, \forall z \in L, < \sigma, z > \notin Y \} \cup$$

$$\{ < \sigma, y > \mid < \sigma, y > \in Y, \forall z \in L, < \sigma, z > \notin X \}$$

Not quite…

Problem:
context length!

Small k =>
imprecise

Solution:
k-limiting

large k =>
poor performance

… but how does it work anyways?

Call Strings
k-limiting

$$
\begin{array}{c}
k() \\
\downarrow \\
C_a \\
\downarrow \\
a()
\end{array}
$$

$(C_0 \bullet C_1 \bullet \ldots \bullet C_k)$

maintain suffix of length k

$(C_1 \bullet \ldots \bullet C_k \bullet C_a)$

Call Strings
k-limiting

- K = length of longest non-recursive call sequence
- $|L|$ = lattice height
- # contexts $= K * (|L| + 1)^2$
- Khedker and Karkare [CC '08] improved this bound to $K * (|L| + 1)$

Call Strings
Recap

- Cloning vs inlining
- Context as string
- Merging call strings
- Context length
- K-limiting

Functional Approach

… but why bother?

Functional Approach

```
  h.f = myPassword();
  i.f = myPhoneNo();
  j.f = myCreditCardNo();

c1 foo(h,x);
c2 foo(i,y);
c3 foo(j,z);

  print(x.g);
  print(y.g);
  print(z.g);
```

```
foo(a,b) {
  c = a.f;
  b.g = c;
}
```

Functional Approach

```
   h.f = myPassword();
   i.f = myPhoneNo();
   j.f = myCreditCardNo();

c1 foo(h,x);                    foo(a,b) {     (a.f, c1)
c2 foo(i,y);                       c = a.f;
c3 foo(j,z);                       b.g = c;    (b.g, c1)
                                }

   print(x.g);
   print(y.g);
   print(z.g);
```

Functional Approach

```
   h.f = myPassword();
   i.f = myPhoneNo();
   j.f = myCreditCardNo();

c1 foo(h,x);                 foo(a,b) {      (a.f, c2)
c2 foo(i,y);                    c = a.f;
c3 foo(j,z);                    b.g = c;     (b.g, c2)
                             }

   print(x.g);
   print(y.g);
   print(z.g);
```

Functional Approach

```
   h.f = myPassword();
   i.f = myPhoneNo();
   j.f = myCreditCardNo();

c1 foo(h,x);
c2 foo(i,y);
c3 foo(j,z);

   print(x.g);
   print(y.g);
   print(z.g);
```

```
foo(a,b) {          (a.f, c3)
  c = a.f;
  b.g = c;          (b.g, c3)
}
```

Functional Approach

```
h.f = myPassword();
i.f = myPhoneNo();
j.
c1
print(x.g);
print(y.g);
print(z.g);
```

Same method analyzed 3 times!!

```
foo(a,b) {
  c = a.f;
  b.g = c;
}
```

(a.f, c1)
(a.f, c2)
(a.f, c3)

(b.g, c3)
(b.g, c2)
(b.g, c1)

Functional Approach
Example: Constant Propagation

```
foo(a,b) {
  a++;
  return a+b;
}
```

Next

- IFDS